Lecture 31-33 - Rootfinding

Table of Contents

Text: Supplementary notes from Instructor

31.1 Why is it important to be able to find roots?

Chemical Engineering is a methodical practice. With the computational tools available today, the biggest difficulty in solving a problem should be properly posing the problem. The actual computations required to extract numeric answers should be trivial. There should be no blood, sweat, or tears required in the numerical solution. Perhaps, for many of you, this is not yet the case. At the end of the course, this should be the case. If you can properly pose the problem, you can solve it.

This section of the course discusses the theoretical basis of some of these numerical techniques. We then apply the techniques to problems typically presented to the chemical engineer. Then we look at how to use a modern computational tool to solve them.

Numerical methods for root-finding have undergone a lot of scrutiny and development with the advent of powerful computers. Computer scientists and mathematicians have teamed up to development powerful algorithms for efficient and accurate root-finding techniques. It is beyond the scope of this course to delve into these state-of-the-art algorithms, used by such software as MATLAB. However, we will discuss some of the basic algorithms, so you understand the fundamental concepts at work.

31.2 Iterative Solutions and Convergence

Our goal is to find the value of x that satisfies the following equation.

$$f(x) = 0 \tag{31.1}$$

where f(x) is some nonlinear algebraic equation. The value of x that satisfies f(x)=0 is called a root of f(x). Therefore, the procedure used to find x is called root-finding.

An example of a nonlinear algebraic equation is

$$f(x) = x - \exp(-x) = 0$$

All root-finding techniques are iterative, meaning we make a guess and we keep updating our guess based on the value of f(x). We stop updating our guess when we meet some specified convergence criterion. The convergence criteria can be on x or on f(x).

$$err_{f,i} = |f(x_i)| \tag{31.2}$$

Since f(x) goes to zero at the root, the absolute value of f(x) is an indication of how close we are to the root.

Alternatively we can specify a convergence criteria on x itself. In this case, there are two types of convergence criteria, absolute and relative. The absolute error is given by

$$err_{x,i} = |x_i - x_{i-1}| \tag{31.3}$$

This error has units of x. If we want a relative error, then we use:

$$err_{x,i} = \left|\frac{x_i - x_{i-1}}{x_i}\right| \tag{31.4}$$

This gives us a percent error based on our current value of x. This error will work unless the root is at zero.

Regardless of what choice of error we use, we have to specify a tolerance. The tolerance tells us the maximum allowable error. We stop the iterations when the error is less than the tolerance.

31.3 Successive Approximations

A primitive technique to solve for the roots of f(x) is called successive approximation. In successive approximation, we rearrange the equation so that we isolate x on left-hand side. So for the example f(x) given below

$$f(x) = x - \exp(-x) = 0$$

we rearrange as:

$$x = \exp(-x)$$

We then make an initial guess for x, plug it into the right hand side and see if it equals our guess. If it does not, we take the new value of the right-hand side of the equation and use that for x. We continue until our guess gives us the same answer.

Successive Approximations Example 1:

Let's find the root to the nonlinear algebraic equation given above using successive approximations. We will use an initial guess of 0.5. We will use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}.

iteration	x	exp(-x)	relative error
1	0.5000000	0.6065307	1.0000E+02
2	0.6065307	0.5452392	1.1241E-01
3	0.5452392	0.5797031	5.9451E-02
4	0.5797031	0.5600646	3.5065E-02
5	0.5600646	0.5711721	1.9447E-02
6	0.5711721	0.5648629	1.1169E-02
7	0.5648629	0.5684380	6.2893E-03

8	0.5684380	0.5664095	3.5815E-03
9	0.5664095	0.5675596	2.0265E-03
10	0.5675596	0.5669072	1.1508E-03
11	0.5669072	0.5672772	6.5221E-04
12	0.5672772	0.5670674	3.7005E-04
13	0.5670674	0.5671864	2.0982E-04
14	0.5671864	0.5671189	1.1902E-04
15	0.5671189	0.5671571	6.7494E-05
16	0.5671571	0.5671354	3.8280E-05
17	0.5671354	0.5671477	2.1710E-05
18	0.5671477	0.5671408	1.2313E-05
19	0.5671408	0.5671447	6.9830E-06
20	0.5671447	0.5671425	3.9604E-06
21	0.5671425	0.5671438	2.2461E-06
22	0.5671438	0.5671430	1.2739E-06
23	0.5671430	0.5671434	7.2246E-07

So we have converged to a final answer of x = 0.567143.

Successive Approximations Example 2:

Now let's try to solve an analogous problem. This problem has the same root as the one we just solved.

$$f(x) = x + \ln(x) = 0$$

we rearrange as:

$$x = -\ln(x)$$

iteration	x	-ln(x)	relative error
1	0.5000000	0.6931472	1.0000E+02
2	0.6931472	0.3665129	8.9119E-01
3	0.3665129	1.0037220	6.3485E-01
4	1.0037220	-0.0037146	2.7121E+02
5	-0.0037146	Does Not Exist	

By iteration 5, we see that we are trying to take the natural log of a negative number, which does not exist. The program crashes.

This illustrates several key points about successive approximations:

Successive Approximation	
Advantages	• simple to understand and use
Disadvantages	• no guarantee of convergence • very slow convergence • need a good initial guess for convergence

My advice is to never use successive approximations. As a root-finding method it is completely unreliable. The only reason I present it here is to try to convince you that you should not use it.

A Matlab code which implements successive approximation is given below. Use it at your own risk.

Since I have included a code, a few comments are necessary.

The code is stored in a file called *succ_app.m*
The code is executed by typing *succ_app(x)* where x is the numeric value of your initial guess.

In the code we first specify the maximum number of iterations (*maxit*), the tolerance (*tol*). We initialize the error to a high value (*err*). We initialize the iteration counter (*icount*) to zero. We initialize *xold* to our initial guess. Then we have a while-loop that executes as long as *icount* is less than the maximum number of iterations and the error is greater than the tolerance. At each iteration, we evaluate *xnew* and the error. On the first iteration, we do not calculate the error, since it takes two iterations to generate a relative error.

If the code converges, it stores *xnew* as *x0* and reports that value to the user. If the maximum number of iterations is exceeded, then we have not converged and the code let's us know that.

The right-hand side of the equation that we want to solve is given as the last line in the code.

```
%
%  successive approximations
%
function [x0,err] = succ_app(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold = x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   xnew = funkeval(xold);
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xnew = %e xold = %e err = %e \n',icount, xnew, xold, err);
   xold = xnew;
end
%
x0 = xnew;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
   fprintf(1,'Maybe you should try a better method because Successive Approximations is lame. \n');
end


function x = funkeval(x0)
x = exp(-x0);
```

31.4 Bisection Method of Rootfinding

Another primitive method for finding roots which we will never use in practice, but which you should be exposed to once is called the bisection method. In the bisection method we still want to find the root to

$$f(x) = 0 \qquad (31.1)$$

We do so by finding a value of x, namely x_+, where $f(x) > 0$ and a second value of x, namely x_-, where $f(x) < 0$. These two values of x are called brackets. If we have brackets, then we know that the value of x for which $f(x) = 0$ lies somewhere between the two brackets.

In the bisection method, we first must find the brackets. After we have the brackets, we then find the value of x midway between the brackets.

$$x_{mid} = \frac{x_+ + x_-}{2}$$

If $f(x_{mid}) > 0$, then we replace x_+ with x_{mid}, namely $x_+ = x_{mid}$. The other possibility is that $f(x_{mid}) < 0$, in which case $x_- = x_{mid}$.

With our new brackets, we find the new midpoint and continue the procedure until we have reached the desired tolerance.

Bisection Method Example 1:

Let's solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0$$

We will take as our brackets,

$x_- = 0.1$ where $f(x_-) = -2.203 < 0$

$x_+ = 1.0$ where $f(x_+) = 1.0 > 0$

How did we find these brackets? Trial and error or we plot f(x) vs x and get some ideas where the function is positive and negative.

We will use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}.

iteration	x_-	x_+	$f(x_-)$	$f(x_+)$	error
1	5.500000E-01	1.000000E+00	-4.783700E-02	1.000000E+00	4.500000E-01
2	5.500000E-01	7.750000E-01	-4.783700E-02	5.201078E-01	2.903226E-01
3	5.500000E-01	6.625000E-01	-4.783700E-02	2.507653E-01	1.698113E-01
4	5.500000E-01	6.062500E-01	-4.783700E-02	1.057872E-01	9.278351E-02
5	5.500000E-01	5.781250E-01	-4.783700E-02	3.015983E-02	4.864865E-02
6	5.640625E-01	5.781250E-01	-8.527718E-03	3.015983E-02	2.432432E-02
7	5.640625E-01	5.710938E-01	-8.527718E-03	1.089185E-02	1.231190E-02
8	5.640625E-01	5.675781E-01	-8.527718E-03	1.201251E-03	6.194081E-03
9	5.658203E-01	5.675781E-01	-3.658408E-03	1.201251E-03	3.097041E-03
10	5.666992E-01	5.675781E-01	-1.227376E-03	1.201251E-03	1.548520E-03
11	5.671387E-01	5.675781E-01	-1.276207E-05	1.201251E-03	7.742602E-04
12	5.671387E-01	5.673584E-01	-1.276207E-05	5.943195E-04	3.872800E-04
13	5.671387E-01	5.672485E-01	-1.276207E-05	2.907975E-04	1.936775E-04

```
14  5.671387E-01   5.671936E-01  -1.276207E-05   1.390224E-04  9.684813E-05
15  5.671387E-01   5.671661E-01  -1.276207E-05   6.313133E-05  4.842641E-05
16  5.671387E-01   5.671524E-01  -1.276207E-05   2.518492E-05  2.421379E-05
17  5.671387E-01   5.671455E-01  -1.276207E-05   6.211497E-06  1.210704E-05
18  5.671421E-01   5.671455E-01  -3.275270E-06   6.211497E-06  6.053521E-06
19  5.671421E-01   5.671438E-01  -3.275270E-06   1.468118E-06  3.026770E-06
20  5.671430E-01   5.671438E-01  -9.035750E-07   1.468118E-06  1.513385E-06
21  5.671430E-01   5.671434E-01  -9.035750E-07   2.822717E-07  7.566930E-07
```

So we have converged to a final answer of x = 0.567143.

This example illustrates several key points about successive approximations:

Bisection Method	
Advantages	• simple to understand and use • guaranteed convergence, if you can find brackets
Disadvantages	• must first find brackets (i.e., you need a good initial guess of where the solution is) • very slow convergence

A Matlab code which implements bisection method is given below.

Since I have included a code, a few comments are necessary.

The code is stored in a file called *bisection.m*
The code is executed by typing *bisection(xn,xp)* where xn is the numeric value of x which gives a negative value of f(x) and xp is the numeric value of x which gives a positive value of f(x). You must give the code 2 arguments for it to run properly. Additionally, these two values must be legitimate brackets.

The code follows the same outline as the successive approximations code. See comments for that code.

The equation, f(x), that we want to find the roots for, is given as the last line in the code.

```
%
%  bisection method
%
function [x0,err] = bisection(xn,xp);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
fn = funkeval(xn);
fp = funkeval(xp);
while (err > tol & icount <= maxit)
   icount = icount + 1;
   xmid = (xn + xp)/2;
   fmid = funkeval(xmid);
   if (fmid > 0)
      fp = fmid;
      xp = xmid;
   else
      fn = fmid;
      xn = xmid;
   end
   err = abs((xp - xn)/xp);
   fprintf(1,'icount = %i xn = %e xp = %e fn = %e fp = %e err = %e \n',icount, xn, xp, fn, fp, err);
end
%
x0 = xmid;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
   fprintf(1,'Maybe you should try a better method because Bisection Method is slow as a dog. \n');
end


function f = funkeval(x)
f = x + log(x);
```

31.5 Single Variable Newton-Raphson - Theory

One of the most useful root-finding techniques is called the Newton-Raphson method. The Newton-Raphson technique allows you to find solutions to a problem of the form:

$$f(x) = 0 \tag{31.1}$$

where $f(x)$ can be any function. The basis of the Newton-Raphson method lies in the fact that we can approximate the derivative of $f(x)$ numerically.

$$f'(x_1) = \frac{df(x_1)}{dx} \approx \frac{f(x_1) - f(x_2)}{x_1 - x_2} \tag{31.2}$$

Now, initially, we don't know the roots of $f(x)$. Let's say our initial guess is x_1. Let's say that we want x_2 to be a solution to $f(x) = 0$. Let's rearrange the equation to solve for x_2.

$$x_2 = x_1 - \frac{f(x_1) - f(x_2)}{f'(x_1)} \tag{31.3}$$

Now, if x_2 is a solution to $f(x) = 0$, then $f(x_2) = 0$ and the equation becomes:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{31.4}$$

This is the Newton-Raphson Method. Based on the x_1, $f(x_1)$, $f'(x_1)$, we estimate the root to be at x_2. Of course, this is just an estimate. The root will not actually be at x_2. Therefore, we can do the Newton-Raphson Method again, a second iteration.

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} \tag{31.5}$$

In fact, we repeat these iteration using the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{31.5}$$

until the difference between x_{i+1} and x_i is small enough to satisfy us.

The Newton-Raphson method requires you to calculate the first derivative of the equation, $f'(x)$. Sometimes this is a hassle. Additionally, we see from the equation above that when the derivative is zero, the Newton-Raphson method fails, because we divide by the derivative. This is a weakness of the method.

However because we go to the trouble to give the Newton-Raphson method the extra information about the function contained in the derivative, it will converge must faster than the previous methods. We will see this demonstrated in the following example.

Finally, as with any root-finding method, the Newton-Raphson method requires a good initial guess of the root.

Newton Raphson Example 1:

Let's solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0$$

The derivative is

$$f'(x) = 1 + \frac{1}{x}$$

We will use an initial guess of 0.5. We will use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}.

iteration	x_{old}	$f(x_{old})$	$f'(x_{old})$	x_{new}	error
1	5.000000E-01	-1.931472E-01	3.000000E+00	5.643824E-01	1.000000E+02
2	5.643824E-01	-7.640861E-03	2.771848E+00	5.671390E-01	4.860527E-03
3	5.671390E-01	-1.188933E-05	2.763236E+00	5.671433E-01	7.586591E-06
4	5.671433E-01	-2.877842E-11	2.763223E+00	5.671433E-01	1.836358E-11

So we converged to 0.5671433 in only four iterations. We see that for the last three iterations, the error dropped quadratically. By that we mean

$$err_{i+1} \approx err_i^2 \text{ or } \frac{err_{i+1}}{err_i^2} \approx 1$$

Quadratic convergence is a rule of thumb. The ratio ought to be on the order of 1.

$$\frac{err_3}{err_2^2} = \frac{7.6 \cdot 10^{-6}}{\left(4.9 \cdot 10^{-3}\right)^2} = 0.32 \qquad \text{and} \qquad \frac{err_4}{err_3^2} = \frac{1.83 \cdot 10^{-11}}{\left(7.6 \cdot 10^{-6}\right)^2} = 0.32$$

This example illustrates several key points about successive approximations:

Newton-Raphson Method	
Advantages	• simple to understand and use • quadratic (fast) convergence, near the root
Disadvantages	• have to calculate analytical form of derivative • blows up when derivative is zero. • need a good initial guess for convergence

A Matlab code which implements Newton-Raphson method is given below.

Since I have included a code, a few comments are necessary.

The code is stored in a file called *newraph.m*
The code is executed by typing *newraph(x)* where x is the numeric value of your initial guess of x.

The code follows the same outline as the successive approximations code. See comments for that code.

The equations for f(x) and dfdx(x) are given near the bottom of the code.

```
%
%  Newton-Raphson method
%
function [x0,err] = newraph(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   f = funkeval(xold);
   df = dfunkeval(xold);
   xnew = xold - f/df;
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e  err = %e \n',icount, xold, f, df, xnew, err);
   xold = xnew;
end
%
x0 = xnew;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end


function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x)
df = 1 + 1/x;
```

31.6 Single Variable Newton-Raphson - Numerical Approximation to Derivatives

People don't like to take derivatives. Because they don't like to, they don't want to use the Newton-Raphson method.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad (31.5)$$

So, if we are one of those people, we can approximate the derivative at x_i using a centered finite difference formula:

$$f'(x_i) = \frac{f(x_i + h) - f(x_i - h)}{2h}$$

where h is some small number. Generally I define h as

$$h = \min(0.01 \cdot x_i, 0.01)$$

This is just a rule of thumb that I made up that seems to work 95% of the time. Using this approximation, we execute the Newton-Raphson algorithm in precisely the same way, except we never to have to evaluate the derivative.

Newton Raphson with Numerical Approximation to the Derivative Example 1:

Let's solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0$$

We will use an initial guess of 0.5. We will use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}.

iteration	x_{old}	$f(x_{old})$	$f'(x_{old})$	x_{new}	error
1	5.000000E-01	-1.931472E-01	3.000067E+00	5.643810E-01	1.000000E+02
2	5.643810E-01	-7.644827E-03	2.771912E+00	5.671389E-01	4.862939E-03
3	5.671389E-01	-1.206407E-05	2.763295E+00	5.671433E-01	7.697929E-06
4	5.671433E-01	-2.862443E-10	2.763282E+00	5.671433E-01	1.826496E-10

So we converged to 0.5671433 in only four iterations, just as it did in the rigorous Newton-Raphson method. This example illustrates several key points about successive approximations:

Newton-Raphson with Numerical Approximation of the Derivative Method	
Advantages	• simple to understand and use • quadratic (fast) convergence, near the root
Disadvantages	• blows up when derivative is zero. • need a good initial guess for convergence

A Matlab code which implements Newton-Raphson method is given below.

Since I have included a code, a few comments are necessary. The code is stored in a file called *newraph_nd.m* The code is executed by typing *newraph_nd(x)* where x is the numeric value of your initial guess of x.

The code follows the same outline as the Newton Raphson code. See comments for that code. The equations for f(x) and dfdx(x) are given near the bottom of the code.

```
%
%  Newton-Raphson method with numerical approximations to the derivative.
%
function [x0,err] = newraph_nd(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   f = funkeval(xold);
   h = min(0.01*xold,0.01);
   df = dfunkeval(xold,h);
   xnew = xold - f/df;
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e  err = %e \n',icount, xold, f, df, xnew, err);
   %fprintf(1,'%i %e %e %e %e %e \n',icount, xold, f, df, xnew, err);
   xold = xnew;
end
%
x0 = xnew;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end


function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);
```

31.7 MATLAB fzero.m - Search and Interpolate

Matlab has an intrinsic function to find the root of a single nonlinear algebraic equation. The routine is called *fzero.m*. You can access help on it by typing *help fzero* at the Matlab command line prompt. You can also access the fzero.m file itself and examine the code line by line.

I am not going to explain the code here, but I am going to outline how the technique finds a solution. The generic technique that the fzero.m function uses to find a solution is classified as a "search and interpolation" technique.

In a "search and interpolation" method, you start with an initial guess to the equation

$$f(x) = 0$$

You evaluate the function at the initial guess. If the function is positive, your initial guess becomes your positive bracket, just as in the bisection case. If the function is negative, your initial guess becomes your negative bracket.

Next, you start searching around the initial guess for the other bracket. The steps might be something of along the lines of

$$\begin{aligned} x_1 &= x_0 + h \\ x_2 &= x_0 - h \\ x_3 &= x_0 + 2h \\ x_4 &= x_0 - 2h \end{aligned}$$

and so forth. At each step, you evaluate the function until you find a value of x which gives you a functional evaluation which is opposite in sign to that of your initial guess.

At that point, you begin a bisection-like convergence criteria, except that instead of using the midpoint as your new guess, you use a linear interpolation between the two brackets.

$$x_{new} = x_- - \frac{f(x_-)}{f(x_-) - f(x_+)}(x_- - x_+)$$

You replace whichever bracket ought to be replaced with x_{new}, based on the sign of $f(x_{new})$. You repeat this procedure until you converge to the desired tolerance.

The actual Matlab code is a little more sophisticated but we now understand the gist behind a "search and interpolate" method.

The simplest syntax for using the fzero.m code is to type at the command line prompt:

```
>> x = fzero('f(x)',x0,tol)
```

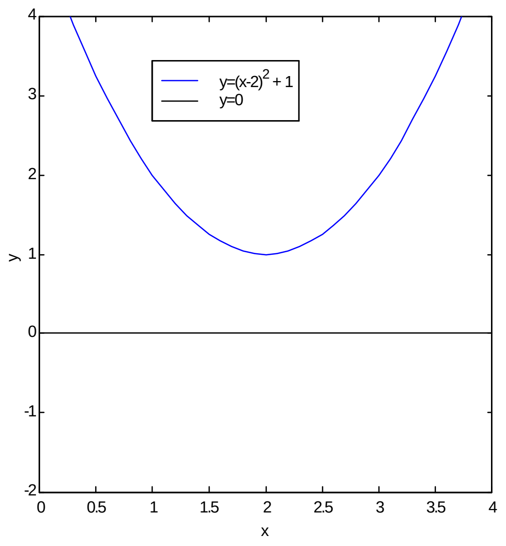
where f(x) is the function we want the roots of, x0 is the initial guess, and tol is the tolerance. An example:

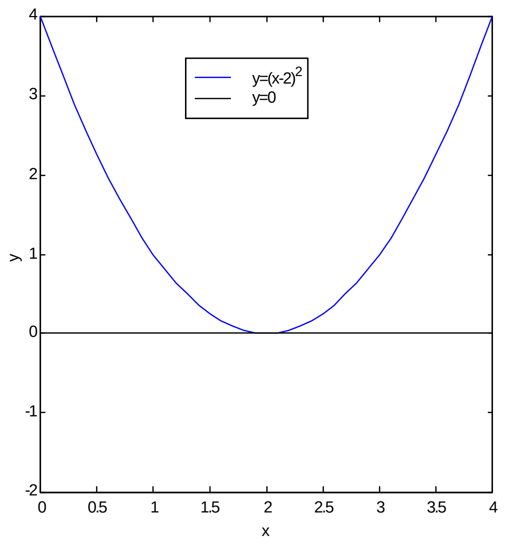
```
>> x = fzero('x+log(x)',0.5,1.e-6)
x =  0.56714332272548
```

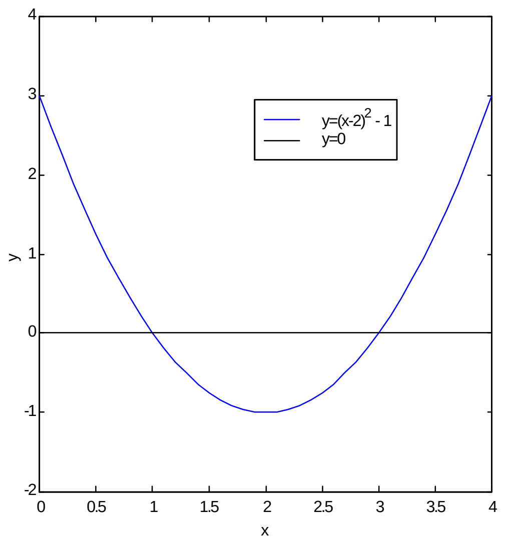
Matlab's fzero.m (search and interpolate)	
Advantages	• comes with Matlab • slow convergence
Disadvantages	• has to find brackets before it can begin converging • need a good initial guess for convergence • somewhat difficult to use. The help file is inadequate.

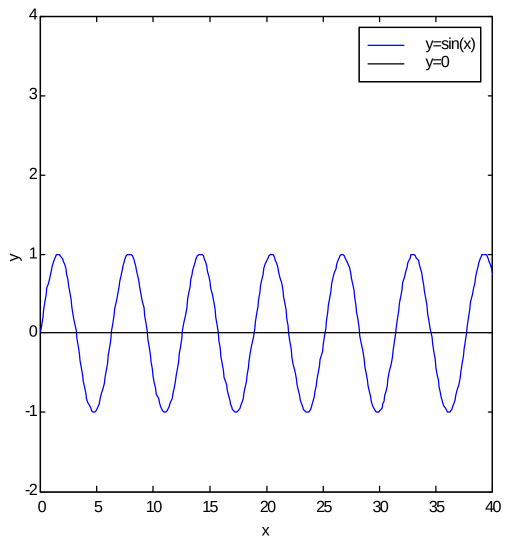
31.8 How Many Roots do nonlinear equations have and how do we find them?

When dealing with linear algebraic equations, we could determine how many roots were possible and there were always 0, 1, or an infinite number. When dealing with nonlinear equations, we have no such theory. A nonlinear equation can have 0, 1, 2… up through an infinite number of roots. There is no sure way to tell except by plotting it out. Here are a few examples of nonlinear equations with 0, 1, 2 and an infinite number of roots.



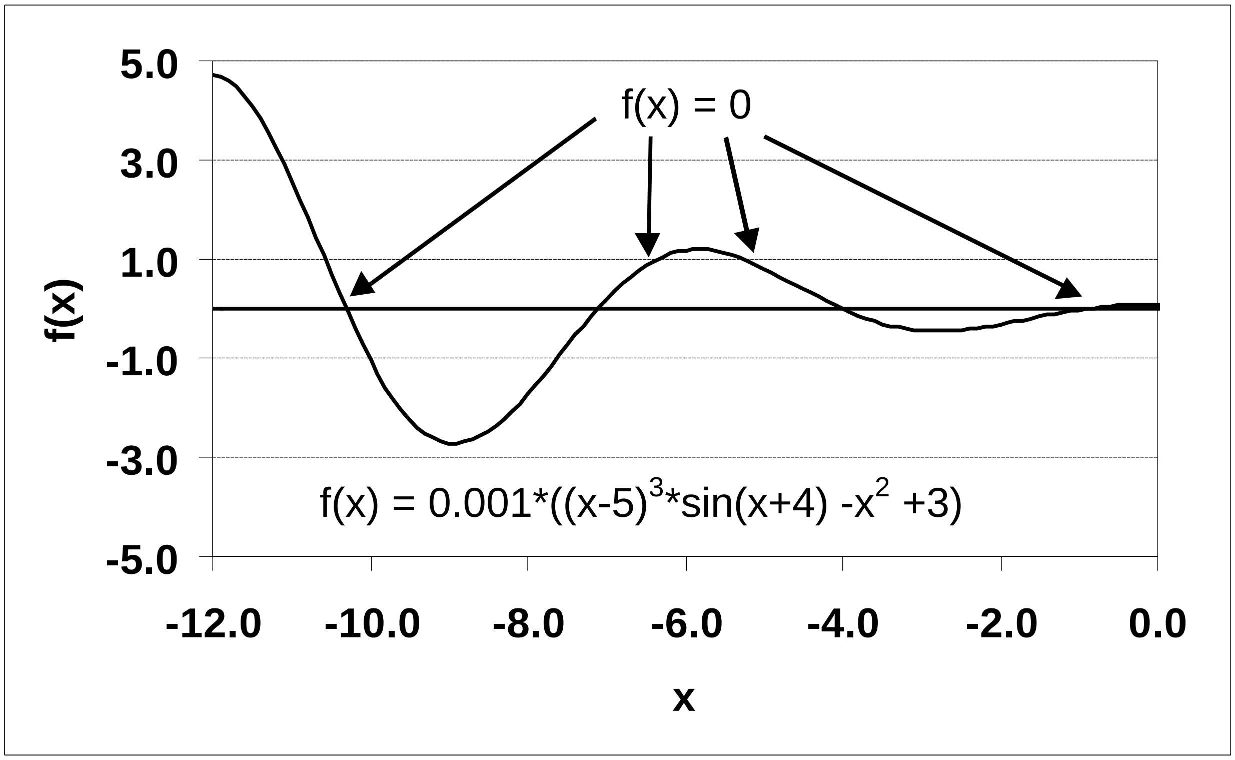






When you use any of the numerical root-finding techniques described above, you will only find one root at a time. Which root you locate depends upon your choice of method and the initial guess.

Example:
Let's find the four roots in the range -12.0 to 0.0 for the function depicted in the plot below, using the Newton-Raphson method.



$$f(x) = 0.001*((x-5)^3 * \sin(x+4) - x^2 + 3) \tag{31.9}$$

The derivative of this function is

$$f'(x) = 0.001\left[(x-5)^3 \cos(x+4) + 3(x-5)^2 \sin(x+4) - 2x\right] \tag{31.10}$$

In an attempt to find all four roots, let's start with the one near x = -10.0 Our iterations will be logged in the following table.

iteration	x_i	$f(x_i)$	$f'(x_i)$	e_i
1	-10	-1.04003	-2.57246	
2	-10.4043	0.336356	-2.90107	-0.40429
3	-10.2884	-0.08439	-2.85159	0.115942
4	-10.3179	0.02145	-2.86789	-0.02959
5	-10.3105	-0.00541	-2.86401	0.007479
6	-10.3124	0.001366	-2.865	-0.00189
7	-10.3119	-0.00034	-2.86475	0.000477
8	-10.312	8.71E-05	-2.86482	-0.00012
9	-10.312	-2.2E-05	-2.8648	3.04E-05

Therefore, one of the roots of $f(x)$ in the range between -12.0 and 0.0 is about -10.312.

The second root in the plot of $f(x)$, looks like it's about -7.0. Let's use the same procedure to find that root.

iteration	x_i	$f(x_i)$	$f'(x_i)$	e_i
1	-7	0.197855	1.29703	
2	-7.15254	-0.06781	1.365912	-0.15254
3	-7.1029	0.021133	1.346601	0.049648
4	-7.11859	-0.00674	1.353041	-0.01569
5	-7.11361	0.002132	1.35103	0.004981
6	-7.11519	-0.00068	1.351671	-0.00158
7	-7.11469	0.000214	1.351468	0.0005
8	-7.11485	-6.8E-05	1.351532	-0.00016
9	-7.1148	2.15E-05	1.351512	5.03E-05

Therefore, one of the roots of $f(x)$ in the range between -12.0 and 0.0 is about -7.1148.

The third root in the plot of $f(x)$, looks like it's about -4.0. Let's use the same procedure to find that root.

iteration	x_i	$f(x_i)$	$f'(x_i)$	e_i
1	-4	-0.013	-0.478	
2	-4.0272	0.006786	-0.48292	-0.0272
3	-4.01314	-0.00348	-0.48042	0.014052
4	-4.02039	0.001801	-0.48172	-0.00725
5	-4.01665	-0.00093	-0.48105	0.003739
6	-4.01858	0.000479	-0.4814	-0.00193
7	-4.01758	-0.00025	-0.48122	0.000995
8	-4.0181	0.000127	-0.48131	-0.00051
9	-4.01783	-6.6E-05	-0.48126	0.000265

Therefore, one of the roots of $f(x)$ in the range between -12.0 and 0.0 is about -4.01783.

The fourth root in the plot of $f(x)$, looks like it's about -1.0. Let's use the same procedure to find that root.

iteration	x_i	$f(x_i)$	$f'(x_i)$	e_i
1	-1	-0.02848	0.108919	
2	-0.7385	0.025059	0.09101	0.261496
3	-1.01384	-0.0317	0.109713	-0.27534
4	-0.72492	0.027447	0.089963	0.288929
5	-1.03001	-0.0355	0.110616	-0.3051
6	-0.70906	0.030184	0.088733	0.320948
7	-1.04923	-0.04008	0.111654	-0.34016
8	-0.69022	0.033365	0.087257	0.359008
9	-1.0726	-0.04575	0.112862	-0.38238
10	-0.66724	0.037139	0.08544	0.405353
11	-1.10192	-0.05299	0.114292	-0.43468
12	-0.63824	0.041736	0.083121	0.463676
13	-1.14036	-0.06271	0.116009	-0.50211
14	-0.59976	0.047553	0.080009	0.540601

15	-1.1941	-0.07671	0.118096	-0.59435

Very quickly we see this doesn't work. It has shot us off to some other root. Why is this the case? Well, from the plot of $f(x)$, we see that there is a root at about x = 1.0. However, if we look at the plot of $f'(x)$, we see that the derivative is close to zero there. Since the derivative is close to zero at that root, the Newton-Raphson Method cannot find it. That's life; sometimes things work, sometimes they don't.

We can repeat this problem in MATLAB, you can use the fzero function. The three inputs are the function, $f(x)$, the initial guess, and the tolerance. Examples using the above function:

```
x = fzero('0.001*((x-5)^3*sin(x+4) - x^2 + 3)',-10,1.e-6)
x =  -10.3120
```

Changing only the second input, the initial guess, we can arrive at the data in this table:

initial guess	final answer
-11	-10.3120
-10	-10.3120
-9	-10.3120
-8	-7.1148
-7	-7.1148
-6	-7.1148
-5	-4.0179
-4	-4.0179
-3	-4.0179
-2	-0.8695
-1	-0.8695
0	-0.8695

MATLAB is using a different technique than Newton-Raphson. This method can find the root near -1.0.

We can get an idea for what sort of algorithm MATLAB is using by including a fourth input in the fzero function. This gives us a trace of the fzero solution path.

```
x = fzero('0.001*((x-5)^3*sin(x+4) - x^2 + 3)',-10,1.e-6,1)
 Func evals      x           f(x)         Procedure
    1           -10      -1.04003        initial
    2       -9.71716      -1.80091        search
    3       -10.2828      -0.10396        search
    4           -9.6      -2.05375        search
    5          -10.4      0.320508        search

   Looking for a zero in the interval [-9.6, -10.4]

    6        -10.292    -0.0713853        interpolation
    7       -10.3117   -0.00106494        interpolation
    8        -10.312  4.04806e-007        interpolation
    9        -10.312 -7.35963e-005        interpolation

x = -10.3120
```

MATLAB scans the local area around our initial guess until it finds a change in sign for $f(x)$. Once it finds that change in sign, MATLAB uses an iterative interpolation scheme to find the solution. This scheme is based on the method of bisection which works if you have an upper and lower bound for the solution, which you do when, you have $f(x)$ positive at one point ($x = 10.4$) and negative at another point ($x = 10.0$). Then $f(x) = 0$, somewhere between them.

One more thing about the Newton Raphson Method. In the above example, the equation was non-linear. If the equation is linear, the NRM takes one iteration to converge to the exact solution. For example,

$$f(x) = 3x + 2 = 0 \tag{31.11}$$

$$f'(x) = 3 \tag{31.12}$$

Let's have an initial guess of 0.

iteration	x_i	$f(x_i)$	$f'(x_i)$	e_i
1	0	2	3	
2	-0.66667	0	3	-0.66667
3	-0.66667	0	3	0

The first value of x after our initial guess is the exact answer. The NRM solves linear equations exactly in one iteration.

31.9 Single Variable Rootfinding - MATLAB - rootfinder.m

On the website, there is a code called rootfinder.m, written by Dr. Keffer at UTK. This code will combine the rootfinding techniques of the intrinsic MATLAB function, fzero, with the plotting capabilities of MATLAB.

The description for how to use the file can be obtained by opening MATLAB, moving to the directory where you have downloaded the rootfinder.m file, and typing

```
help rootfinder
```

This yields:

```
rootfinder(xo,xstart,xend)
  rootfinder will find the root of a single non-linear
  algebraic equation, given the initial estimate, xo.
  It will plot the function in the range xstart to xend.
  The function must be given in the file fnctn.m
  Any and all of the arguments to rootfinder are optional.

  Author:  David Keffer Date: October 22, 1998
```

So, for example, if you entered, at the MATLAB command line interface,

```
rootfinder(1,0,10)
```

this would find the root nearest your initial guess of 1 and then plot the function from 0 to 10, putting markers at the point of your initial guess as well as at the converged root. If you omit the limits of plotting, then the program chooses some (perhaps unhelpful) default limits.

The equation to be solved by rootfinder must be located in the file fnctn.m. It can be as simple as:

```
function f = fnctn(x)
f = 0.001*((x/4-5)^3*sin(x/4+4) - (x^2)/16 + 3)*exp(-x/40);
```

Or, the file fnctn.m can be more complicated, such as finding the roots of the van der Waal's equation of state.

```
function f = fnctn(x)
%  van der Waal's equation of state
a=0.1381; % m^6/mol^2
b=3.184*10^-5; % m^3/mol
R=8.314;  % J/mol/K
T=98;    % K
P = 101325; % Pa
f = R*T/(x-b)-a/x^2 - P;
```

It doesn't matter how complicated the fnctn.m file is so long as, at the end, you give the value of the function of interest.

An example:
Let's find the three roots (the vapor, liquid, and intermediate molar volume) of the van der Waal's equation of state. The fnctn.m file is given above.

First we find the vapor root. We can get an initial guess for this value from the ideal gas law.

$$\frac{V}{n} = \frac{RT}{P} = \frac{8.314 \cdot 98}{101325} = 0.00804$$

```
>> rootfinder(0.008,1.0e-3,1.0e-1)
ans = 0.00790121184303
```

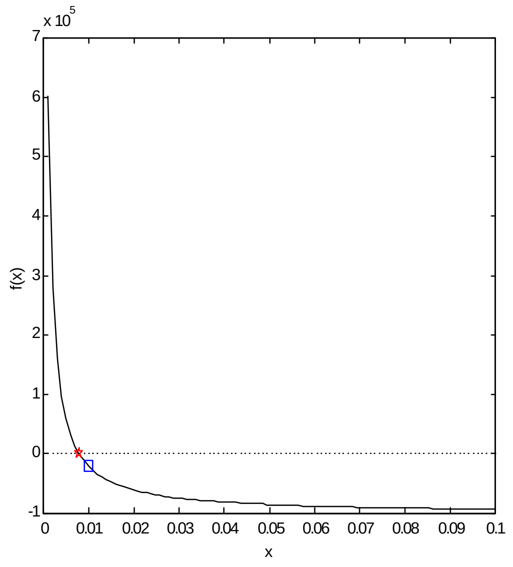
Here the second and third arguments are just lower and upper limits on the x-axis of the plot. In the plot, the blue square is the initial guess and the red star is the root.

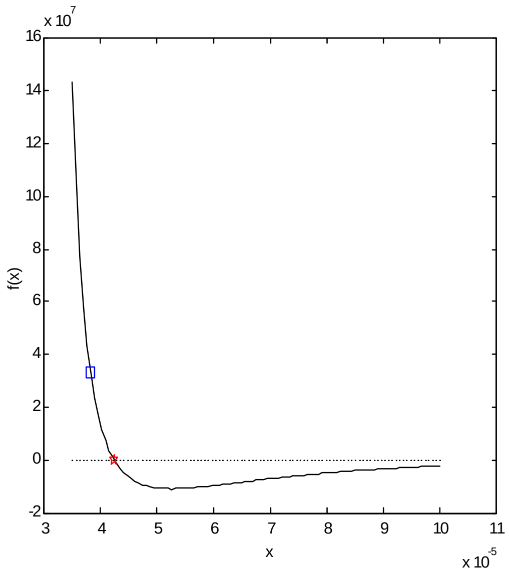
Second, let's find the liquid root, which is probably about 20% larger than the parameter b

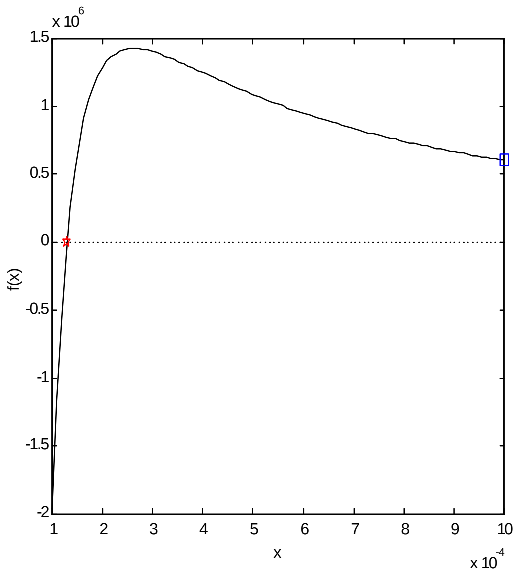
```
» b = 3.184*10^-5;
» rootfinder(1.2*b,1.1*b,1.0e-4)
ans = 4.246507385109214e-005
```

Last, let's find the intermediate roots, which is somewhere between the two roots we have already found.

```
>> rootfinder(1.0e-3,1.0e-4,1.0e-3)
ans = 1.293375218680640e-004
```







Having the plots available helps us locate good initial guesses for the three roots to van der Waal's equation of state.

31.10 Single Variable Rootfinding - MATLAB - GUIs

A GUI (pronounced gooey) is a graphical user interface, which makes using codes simpler. There is a GUI for rootfinding, developed by Dr. Keffer, located at: http://clausius.engr.utk.edu/webresource/index.html .

You have to download and unzip the GUI. Then, when you are in the directory where you extracted the files, you type:

```
>>aesolver1_gui
```

at the command line prompt to start the GUI.

This GUI, titled aesolver1_gui, solves 1 algebraic equation and provides a plot. The purpose of the GUI is to be self-explanatory. I just make a few comments here. The equation to be solved can either be into this GUI directly. You can choose one of four methods to find the root

- fzero.m (search and interpolate)
- Newton Raphson with Numerical Approximations to the Derivative
- Brent's Line Minimization without derivatives
- Brent's Line Minimization with derivatives

These last two methods are taken from Numerical Recipes, the best existing resource for numerical methods. This book happens to be available on-line for free at http://www.ulib.org/webRoot/Books/Numerical_Recipes/ . If you are interested about Brent's line minimization, you can read about it in their book.

Brent's Line Minimization is a minimization routine, not a root-finding technique. A root-finding technique solves

$$f(x) = 0$$

A minimization technique simply finds a local minima of f(x). Generally we think of a local minima as being defined as the x where

$$f'(x) = 0$$

However in these minimization methods, they are bracketing routines (like Bisection) that search for the lowest value of f(x) that they can find.

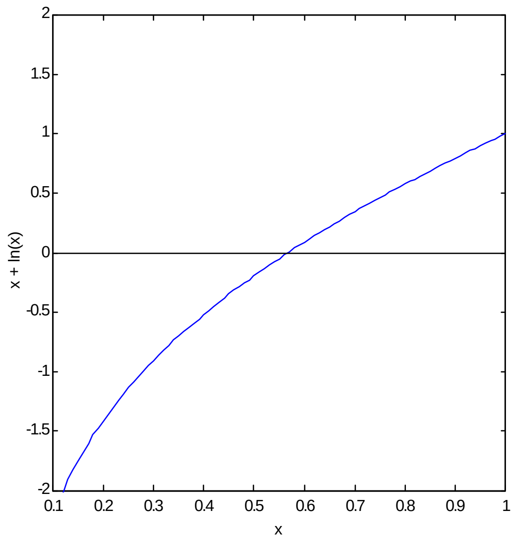
A minimization routine can be used to find the root of a function, if you take the absolute value of the function.

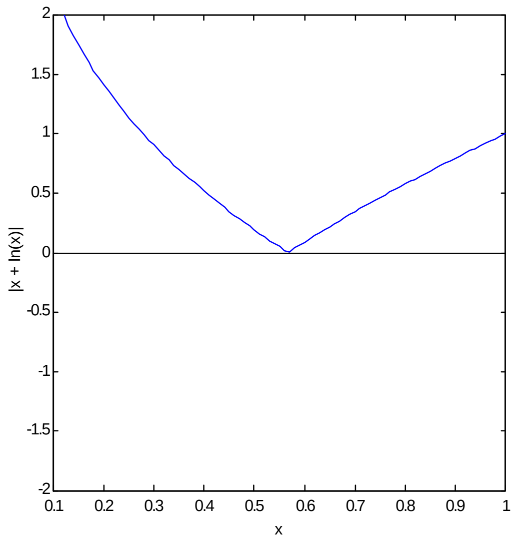
An example: Consider the function

$$f(x) = x + \ln(x)$$

The function and its absolute value are plotted below. Root-finding techniques use the function itself to find the root. A minimization routine uses the absolute value of the function to find the smallest value of the function, which in a root-finding problem is always zero.

The GUI offers multiple methods because each method has strengths and weaknesses. If one of the methods doesn't work, try another one. If none of them work, try an initial guess.





f(x) = x+ ln(x)

f(x) = |x+ ln(x)|

Root-finding on the plot on the left or minimization of the plot on the right, yields the same result, namely the root is located at 0.567154